SPSR

Efficient Processing of Socially k-Nearest Neighbors with Spatial Range Filter

by

Nitin Pasumarthy

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2016 by the
Graduate Supervisory Committee:

Mohamed Sarwat, Chair
Paolo Papotti
Arunabha Sen

ARIZONA STATE UNIVERSITY

August 2016

ABSTRACT

Social media has become popular in the past decade. Facebook for example has 1.59 billion active users monthly. With such massive social networks generating lot of data, everyone is constantly looking for ways of leveraging the knowledge from social networks to make their systems more personalized to their end users. And with rapid increase in the usage of mobile phones and wearables, social media data is being tied to spatial networks. This research document proposes an efficient technique that answers socially k-Nearest Neighbors with Spatial Range Filter. The proposed approach performs a joint search on both the social and spatial domains which radically improves the performance compared to straight forward solutions. The research document proposes a novel index that combines social and spatial indexes. In other words, graph data is stored in an organized manner to filter it based on spatial (region of interest) and social constraints (top-k closest vertices) at query time. That leads to pruning necessary paths during the social graph traversal procedure, and only returns the top-K social close venues. The research document then experimentally proves how the proposed approach outperforms existing baseline approaches by at least three times and also compare how each of our algorithms perform under various conditions on a real geo-social dataset extracted from Yelp.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER 1

INTRODUCTION

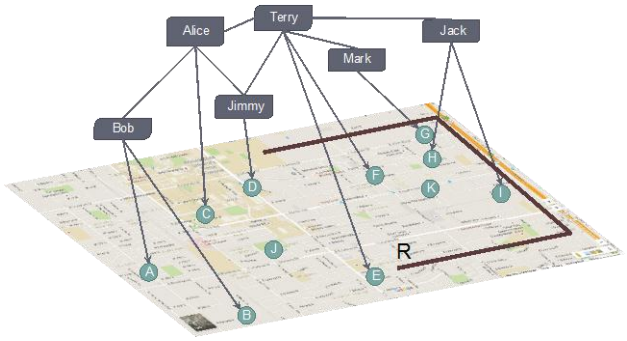

Figure 1. A motivation example showing a socio-spatial graph

A social network is a graph of individuals and their interactions. Users keep up-to date of what their friends like, watch, see, etc... With the ever increasing use of web, social media is also used as an advertising tool. They are proved quite effective [2], [4], [6] to gain quick popularity by publicizing on popular social media sites like Facebook than traditional advertising means. Their effectiveness is mainly due to high usage and size of users using it. Facebook for example has around 1.59 billion active users monthly[1]. Another popular microblogging site called twitter has about 310 million monthly active users[2]. With such massive networks generating lot of data, everyone is constantly looking into ways of integrating the knowledge from them to make their systems more personal for their end users. Microsoft now ranks results, in its BING search, for a user using the search history from his/her social network [9]. There are also works on how probable a user performs an action given his/her friend committed the same action before [10]. The natural problem of social influence would be, 'given a social network, how can we detect the players through which we can spread, or "diffuse", the new technology in the most effective way' [7]. Spread maximizing problem which try to find a minimal set S in a graph to gain maximum spread in a network is well studied in [12].

---

[1] http://www.statista.com/statistics/264810/number-of- monthly-active-facebook-users-worldwide/

[2] http://www.statista.com/statistics/282087/number-of- monthly-active-twitter-users/

At the other end of the spectrum are the spatial networks. With the ever increasing number of wearables everyday like Jawbone, Fitbit, smart watches (pebble, apple watch) there is an abundance of data in this realm too. Importantly with the rapid increase in the number of mobile phone users, this data is already being tied to Social Network data and the two realms are coming together. Popular social network sites like Facebook have a number of features which prompt users to add spatial information like check-ins, traveling posts, geotagged photos etc.

And this research document is all about bringing these two even closer. We can predict what your best friend would have suggested to you if you wanted to go to an authentic Sushi place in SFO. Imagine restaurant recommendations from Google, are more personalized for a location instead of listing them by average user rating. To answer such queries it is required to traverse a social network and filter recommendations which fall in a given region. However minimum latency is rather imperative for such queries for the best user experience using huge social networks like Facebook, Twitter and Yelp. These graphs can be really dense, as much as, each person in the world is connected to every other person by only an average of three and a half other people! [15]. So the way to answer shortest path reachability queries with a spatial predicate quickly is needed even in such dense graphs.

Consider a restaurant recommendation system like Yelp. Every registered user can have multiple friends and also check-ins at multiple venues using this service. A small example from such a service would look like the one shown in Figure 1 where Alice, Bob, Terry, Mark, Jack, Jimmy are people and letters from A to I are venues. Venues are also marked at the respective locations on a map. Edges between people indicate they are friends like in any social network and edges from a person to a restaurant means he/she checked-in at that location. Assume the system wants to recommend a restaurant to Bob in the marked region R and that all venues have the same average customer rating as 4.0. Any existing system would naturally return venues that fall in R in some random order as all of them are equally good. However, Bob is socially close to Terry than to Mark or Jack. Recommending restaurant F before G, H and I would make Bob happier as Terry and Bob are more similar in their tastes. Therefore, in order to provide good recommendations, we should consider both the spatial and social proximities in the search.

An easy way to solve the problem would be to find shortest distances to each venue falling in R, sorting them based on distance from the user of interest and picking the top K (whatever number is required). This disjoint approach can solve the problem but has a huge time latency especially while finding the closest vertices. Also we will be traversing huge graph aimlessly until we hit the required number of venues in R. Such a system would never be used in production. The paper [1] solves the problem in the disjoint manner using a distributed approach by implementing complex algorithms in a bottom up manner using simple distributed functions. This is a good start but we end up with other problems which distributed systems face today like network latency, consistency etc. The paper [11] takes a new approach by combining the social and spatial constraints of the problem during the search routine but is more suited for queries like finding nodes close to a given node. The system in [24] categorizes users as location experts based on their history and uses this precomputed data to generate recommendations for a specific region at runtime. The focus is on using user preference history to generate authorities in a social graph for every region, which is not truly using the user's social network at query time. In our case user's social graph is traversed and top-k recommendations are provided in a given region at runtime which is much more valuable and relevant.

However the aim is to find K closest vertices in a region from a (person) vertex in a given social graph. Here the goal is to process the query with minimum latency and not to propose another recommendation algorithm. The edge weights in the given geosocial graph decide the social distance between two nodes which is used in deciding the social proximity during traversal. So the big challenges ahead are to perform geosocial searches on huge graphs (i) with minimum latency, (ii) traverse the graph in a goal oriented manner towards the region unlike Dijkstra's, (iii) traverse the graph to the minimum as only the K closest vertices to a given vertex are required.

In this research document, we propose a new approach SPSR in order to solve SkNGeo query which finds top-k closest vertices to a given (person) vertex in a social graph considering both the social and spatial components. Our key contributions are as follows:

- Study the geosocial graph problem describing the challenge more formally and understand the need to solve this more efficiently.

- Propose indices on social and spatial domains of the graph which form the pre-processing stage of the solution. Here graph data is stored in an organized manner to filter it based on spatial (region of interest) and social constraints (top-k closest vertices) at query time. This helps in solving the challenge of traversing to the minimum, as only best K are needed.
- Propose a robust algorithm which uses above indices to answer top-k socio-spatial query using a modified landmark based A* algorithm by combining it with a spatial search. This solves the challenge of goal oriented search to reduce the latency even further.
- Experimentally evaluates the proposed approach with different parameter combinations on Yelp
- dataset. The experiments shows that our approach can achieve at least 3 times faster than existing approaches.

CHAPTER 2

PRELIMINARIES

In this chapter all the preliminary information required to understand the problem and solution better is discussed. In particular concepts on what graphs are and some examples on them, an algorithm very widely studied called the shortest path, more detailed study of a type of graph called directed acyclic graph, a term often used in graphs called connected component, a process some graphs undergo called graph condensation.

Besides these concepts, there are two state of the art solutions namely SocialFirst and SpatialFirst, which take a disjointed approach filtering first by social and then spatial or spatial and then social constraints respectively. These are the naïve solutions to solve our problem and are described at the end.

Graph

Graph is another data structure in computer science like Arrays, Linked List, and Trees. A graph contains a finite set of vertices and a finite set of edges connecting them. An edge connects two vertices and can be either directed or undirected. If edges in a graph have a direction (pointed arrow), the graph is called a directed graph or else the graph is undirected graph.

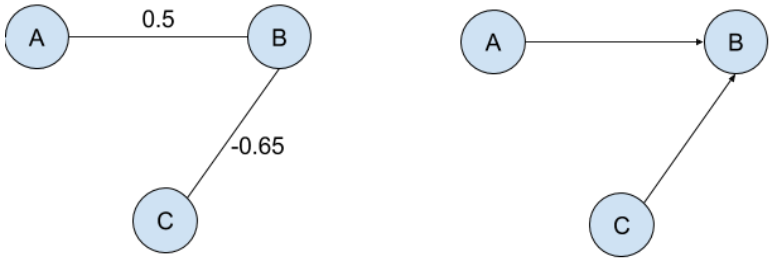

Figure 2. Undirected and Directed Graphs

As shown in Figure 2 the circular shapes are called vertices and lines connecting them are edges. The figure on the left is an undirected graph and one on the right is a directed graph. Both vertices and edges can have additional information attached to them. Here we labelled each vertex as A, B, C etc. and edges with some number on them. Typically the number on the edge is called its weight or importance. Such graphs are called weighted graphs. In our report, we will

always use weighted directed graphs unless stated. We can always convert an unweighted graph to a weighted graph by assuming all weights are same.

Graphs are sometimes referred to as networks. And we can see many examples in our daily life. There are social graphs like Facebook where each user is a vertex and two users are connected by a relationship, generally friendship. The entire web is also a graph where each web page is a vertex and hyperlinks from one web page to another can be directed edges among them. In fact problems are easy to visualize and solve as graph problems. In Biology, protein interaction problems are often studied as graph problems. In such graphs, every protein known is a vertex in the graph and interacting proteins form edges.

Let us now see how graphs are formally represented. There are two ways to represent a graph, G = (V, E) as an adjacency list or adjacency matrix. Here, V is the set of vertices and E is the set of edges which is a list of tuples containing vertices that are connected. If the graph is directed, the first vertex in the tuple is the source, from where the arrow originates, and the second is the destination. In Figure 2, the undirected graph can be represented as V = {A, B, C} and E = {{A, B}, {B, C}} and directed graph is represented as V = {A, B, C} and E = {(A, B), (C, B)}. However, edges are often saved as list or matrix form. In the list form, for every vertex we save a list of vertices it is originating to as shown in Table 1.

Table 1

Adjacency List

Vertex	List
A	B
B	
C	B

In the matrix form for representing edges, we create an N X N matrix where N is the number of vertices. For every edge we mark the corresponding cell in the matrix under the vertices that form the edge. As shown in Table 2, we mark with 1 whenever there is an edge between the vertices else 0. The first row and column are just markers and do not need special space allocation while

coding. Adjacency list occupies less space but access time to check if there is edge between two vertices does not take constant time unlike in adjacency matrix. If the graph is really dense, matrix is a good choice, else list saves space and there are ways to reduce the linear time search by using a hash index. Unless stated, we will use adjacency list for storing edges by default for all algorithms.

Table 2

Adjacency Matrix

	A	B	C
A	0	1	0
B	0	0	0
C	0	1	0

GeoSocial Graph Data

It is a graph contains people, real spatial venues and relationships among these entities. The following is used to model a GeoSocial graph. It is a directed graph G = (V, E, S) consisting of (1) a set of vertices V representing people and spatial venues; (2) a set of directed edges, $E \subset V \times V$ with weights. If $(u, v) \in E$, there exists one edge from vertex u to v, which means the two entities possess a real-life connection. The connection can be Friend-of or Like. The weights of the edge indicates which how strong the two entities are connected. The shorter a distance, the stronger it is; (3) a function S defined on V that decides spatial attribute of a given vertex. S(v) returns spatial property of v (denoted as v.spatial), generally for venues, and the value will be null when v brings no spatial attribute, generally for people. S(v) can be a geometrical a point, line, or polygon. For ease of presentation, we assume that a spatial attribute of spatial vertex is represented by a point.

Shortest Path

In this section we will see one of the graph algorithms called the shortest path or more specifically single-source shortest path. Alice wants to go from her house to Taco Bell in Tempe, Arizona for dinner. She is very hungry and wants to reach as soon as possible. One way would be to enumerate all possible routes from her house to Taco Bell, add up all distances along each route and pick the smallest. Though this approach works, this can be very time consuming. Besides being slow, this won't work, if any of the routes has a cycle. So we need a better way to find a solution to such problems. Given a weighted directed graph, G (V, E) and weight function such that, w(e in E) → **R**, mapping edges to real valued numbers. The weight of a path p(u, v), w(p) = ∑ w(e ∈ E), sum of weights of edges that form the path p. We want the shortest path by weight which is,

$$\beta(u, v) = min(w(p_i) \ \ni p_i \ is \ a \ path \ from \ u \ to \ v) \qquad (1)$$

A shortest path from vertex u to vertex v is defined as the path from u to v which has the least weight compared to all others paths from u to v. The main idea is that every vertex in the shortest path also has the shortest distance to every other vertex in the path. Various approaches exists for different settings of the graph – only positive edge weights, negative weights are allowed but not cycles, only unweighted graph and there can be cycles etc. Breadth first search algorithm finds the shortest path between any two vertices in an unweighted graph with or without cycles. Dijkstra's algorithm finds the shortest path in a weighted graph with or without cycles. This is a greedy algorithm and we will see this algorithm in detail as it is used in one of the solutions to our problem later. Bellman Ford algorithm finds the shortest path even the graph has negative weight edges but it shouldn't have any cycles and is based on dynamic programming. A* algorithm also finds the shortest path between two vertices using a heuristic based approach. We will see this algorithm also in detail later.

Directed Acyclic Graph (DAG)

Directed Acyclic Graph or a dag for short is a graph in which linear ordering or vertices is possible. For every edge (u, v), u appears before v in the linearly ordered list. This is another way

to say that the graph doesn't have a cycle. DAGs are used in many applications mainly because of its acyclic property. In our application, we will first transform a given graph into a DAG and simplify the pre-processing. Figure 3 shows an example DAG and we can arrange the vertices in a linear order. Using topological sort we can arrange a DAG's vertices in order.

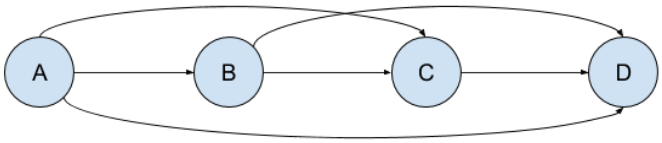

Figure 3. A directed acyclic graph

Strongly Connected Component

A strongly connected component in a directed graph G(V, E) is a maximal set of vertices in G, such that there exists a path between any two vertices. More formally, for every pair of vertices u and v in that set, there exists u → v and v → u. Such a set is called a strongly connected component. It is also good to know that if there exists a path to reach every vertex in that set, it is called a weakly connected component. More formally for every pair of vertices in a set of vertices there exists u → v or v → u. We can find all strongly connected components in a directed graph by using depth first search on it. This can be done in Θ(V + E) time. A graph which has only one strongly connected component is called a Strongly Connected Graph as shown in Figure 4.

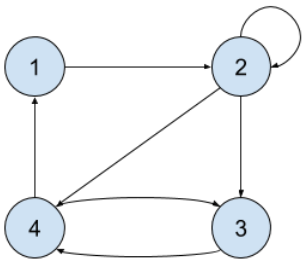

Figure 4. Strongly Connected Graph

Graph Condensation

In a directed graph, if each strongly connected component is contracted into a single vertex, the graph is said to be condensed. Such a graph is always a DAG. The converse, a directed graph is acyclic if every vertex forms its own strongly connected component. This is true as if there are two vertices that form a strongly connected component, then there is a cycle between them. This

can be proved by induction for n vertices. Figure 5 shows how a graph looks before and after condensation. Graph with tiny blue vertices is the original graph and each of its strongly connected component is contracted to one big yellow vertex.

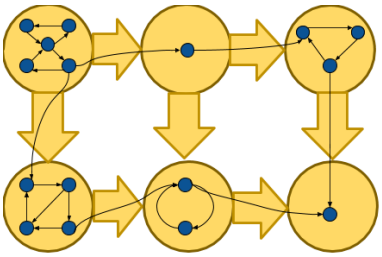

Figure 5. Directed graph with yellow vertices is obtained by contracting the graph with blue vertices

Dijkstra's Single Source Shortest Path Algorithm

Dijkstra's single source shortest path algorithm as the name suggests finds a shortest path from a source vertex to every other vertex in the graph. However, the graph should be weighted and all edge weights should be positive. Bellman-Ford algorithm handles the case when edge weights are negative however, it is more expensive w.r.t running time than Dijkstra's. It uses a minimum priority queue keyed known distances from source vertex, to traverse the graph in order of increasing distance from source.

Algorithm

1. Initialize (G, s)
2. Q = G.V
3. While Q is not empty:
4. u = extract min from Q
5. For each vertex in G.Adj(u):
6. Relax(u, v, w)

Assume every vertex has an attribute called d, which is known distance from the source vertex s. If we also want the shortest path we also maintain parent attribute for each vertex. Line 1 of the algorithm initializes the graph by assigning d = ∞ for every vertex and d = 0 for the source. We also set the parent attribute for every vertex as NULL or unknown. On line 2, we initialize a minimum priority queue Q of vertices of G keyed d attribute of each vertex. Then we pop each vertex from Q and update the distance from the source.

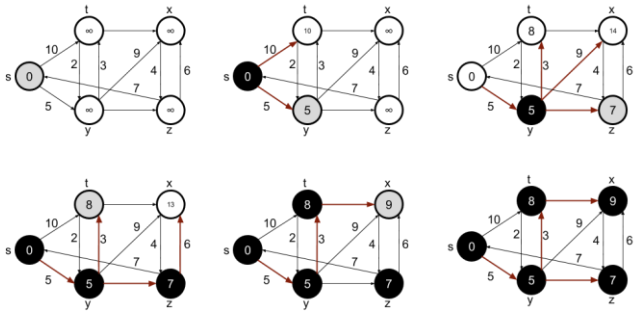

Figure 6. Step by step execution of Dijkstra's Algorithm

Figure 6 shows each execution cycle of while loop on lines 3 to 6. First we pop vertex s as its d is the smallest shown in (a). Due to Relax call on line 6, vertices t and y, the neighbors of s are updated with their actual distances from s. Then out of the unvisited vertices, we pop the least which is y and update its neighbors. Every time we pop a vertex, it is considered to have the smallest distance to source. Here, after (c) the shortest distance to reach y from s would be 5. While reducing neighbors of y, we update the existing distance of t to 8. So instead reaching directly from s, it is shorter w.r.t to weight function to reach t via y. The algorithm stops once all vertices are popped.

The algorithm stops as we are no adding vertices to the Q once they are popped. The invariant is the set of vertices popped till now + set of vertices in the Q = G.V. The runtime of the algorithm is O(E log(V)). In one of our solutions we use Dijkstra's as a starting point and improve upon the ideas for our problem.

A* Single Source Shortest Path Algorithm

Breadth first search (BFS) and Dijkstra's algorithms find the shortest by exploring aimlessly in all directions though we have a single destination. In other words, they are good if we want to find the shortest distances to many destinations. However in our case we have a single source and a single destination. Can we guide the algorithm in the direction of the goal to find the path faster or to traverse the graph lesser? The idea is to use a heuristic function which guides the algorithm. Say we have a *heuristic function that returns an estimated distance to the goal*. Instead keying by

distance from source in Dijkstra's we key by the heuristic distance to goal (or destination). Let us name is greedy breadth first search assuming all edges are of equal weight.

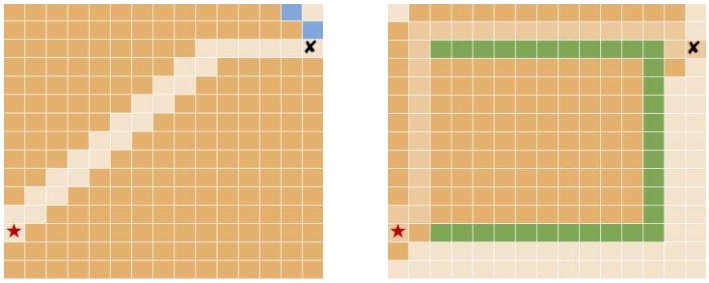

Figure 7. Path finding difference between BFS and Greedy (Heuristic based) BFS

Figure 7 shows another way to visualize a graph, in the form of a grid. Every cell is a vertex and all adjacent cells are connected by edges. Star vertex is the source and cross is our destination. We use early exit version of BFS and Greedy BFS where we stop the traversal once we reach the destination. Blue vertices are vertices in the minimum priority queue and dark brown vertices are vertices explored by the algorithm. Greedy BFS totally beats BFS as it exactly finds the right shortest path to the goal by exploring lesser area of the graph. So the heuristic approach is better for finding paths to a few destinations (and there are proofs for it). However, the celebration ends soon when we add obstacles in the grid, i.e. areas which cannot be traversed, think of them vertices not connected or having edges with ∞ weight. Greedy BFS finds a path much faster like before than BFS but it is not the shortest. As shown in Figure 8, explores less edges but finds a longer path to destination. The dark green area on the grid is the obstacle. The problem lies in the heuristic function where we only account for nearness to the goal.

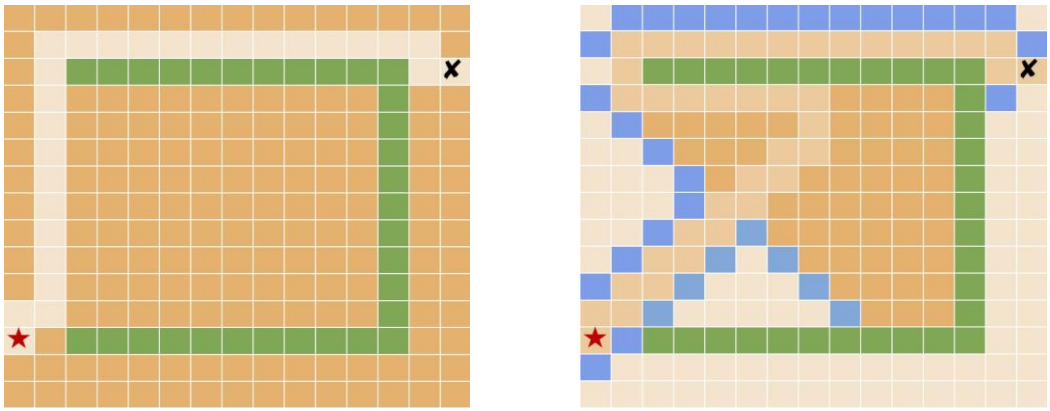

Figure 8. Greedy BFS fails to find the shortest path if there are obstacles

The A* algorithm which was initially written for bot path finding uses best of both the worlds from Dijkstra's and Greedy BFS. The heuristic function is the *sum of actual distance from the start and estimated distance to the goal*.

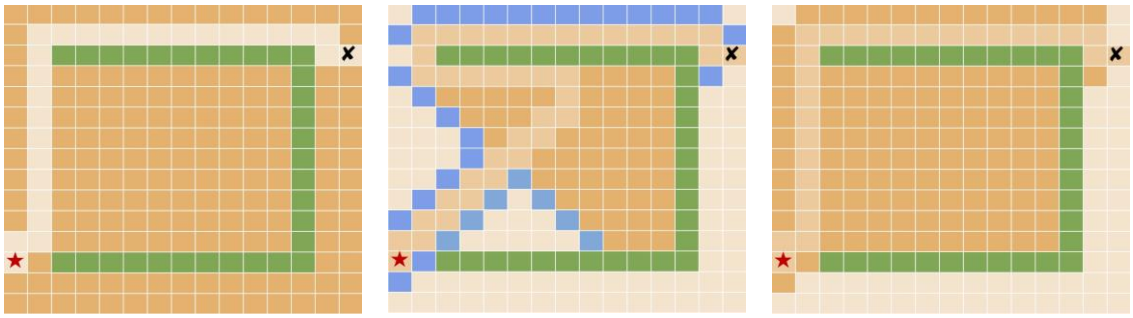

*Figure 9. Comparison among Dijkstra's, Greedy BFS and A**

Figure 9 shows how A* explores only as much as Greedy BFS but also finds the shortest path and not just any path to destination. On the other hand Dijkstra's explores almost the entire graph to find the right path. The only condition for heuristic is it should not overestimate the distance to goal. It uses this heuristic to reorder the nodes in the priority queue, i.e. more intelligence considering the goal.

Social First Algorithm

This algorithm uses the social constraint of K shortest distances first and then checks for spatial predicate. State of the art single source shortest path algorithm traverses the graph greedily from the source vertex until K closest vertices to source in the query region are found. During traversal, every vertex is checked if it falls in the region and the algorithm stops once K vertices are found. As this is a greedy approach which finds vertices in the increasing order of their distances to the source, the first K vertices found in the region are the closest ones. One example of such greedy algorithm can be Dijkstra's.

Spatial First Algorithm

In this algorithm, vertices in the given region R are first filtered and the shortest distances to each are found. After sorting them in ascending by distance from the source, first K are picked. For filtering the vertices in R, state of the art spatial index, r-tree, is used and for finding the shortest

distance from the source vertex to each vertex in the region, state of the art A* with landmark algorithm [8] is used. This algorithm finds the shortest path between two vertices using precomputed landmarks like explained in algorithm 2. The closest K vertices based on distances returned by A* with landmark algorithm are picked.

CHAPTER 3

PROBLEM DEFINITION

In this chapter we will see a detailed explanation to the problem we are solving. Till now we saw a graph defined as G(V, E), from now on a graph is defined as G(V, E, S) where V is still the set of vertices, E is still the set of edges represented in the form of an adjacency list, S is a function defined on a vertex which returns the spatial attribute of it if present else NULL. So S has all information about the vertices that are spatial. Spatial attribute can be latitude and longitude information representing a point, for simplicity in our case.

Path: A path is an ordered list of vertices and the length of the path is defined as the summation of all edge weights along the path.

$$p(u, v) = (v_1, v_2, \dots v_{n-1}, v_n) \quad (2)$$

$$L(p) = \sum w(edges(p)) \quad (3)$$

Single Source Shortest Path: It is a path from a given vertex to a given destination which has the least length. If there are multiple paths from vertex u to vertex v, shortest path is the one which has the least weight.

$$ShortestPath(u, v) = min(L(p_i) | p_i = p(u, v)) \quad (4)$$

Socially k-Nearest Neighbors with GeoSpatial Range Filter(SkNGeo): Given a source vertex and a spatial region, a shortest path is defined as a path with the smallest length from the source vertex to any (spatial) vertex that falls in the region. Here spatial region can be a bounding box on the world map.

SkNGeo gets as input a source vertex v, a spatial region R and number of returned venues k and returns a set that consists of k venues that are located in R and can be reachable from v through k shortest path among all located venues.

Social Proximity & Spatial Reachability(SPSR): returns the top-k nearest vertices from v to R. If we filter the vertices that fall in R and arrange shortest paths to each in ascending order of their

length, Social Proximity & Spatial Reachability (abb. as SPSR) returns the first K in such an ordering. Ultimately, K closest vertices to source vertex in the region are fetched.

To understand how a sample solution looks like, See Figure 1 to understand the problem better. When a query SkNGeo(Terry, R, 2) is issued, {G, F, H, I, K} are all the vertices that are located in the region R. Aim of the query is to find two venues socially closest to Terry among {G, F, H, I, K}. Assume that F and G have shorter distance to Terry than the other vertices {H, I, K (not reachable)}, then G and F will be result of such SkNGeo query.

CHAPTER 4

SOLUTION

The solution to SPSR would be a list of K closest vertices to source vertex that lie in the region. A simple solution would be to first filter all vertices in the region and find shortest distances to each using a single source shortest path algorithm like Dijkstra's or a more robust state of the art A* with landmark algorithm [8]. Once all paths are found we sort them by their length and pick the destination vertices of first K paths. This works but a spatial index like R-Tree would be needed to execute the spatial predicate. Also this needs traversal of the graph until distances to all vertices in R are found in the worst case if we use Dijkstra's. In many case we may just need a K which is much smaller to the number of vertices in R. If A* with landmark algorithm is used to find the path lengths after filtering the vertices in R, it would take a really long time for reasons mentioned in the experiments chapter.

Another idea would be to pre-compute shortest distances from all vertices to all other vertices in the graph. Save these distances and paths in a relational database and index them. On query, the index is probed and K best in the list of filtered vertices by spatial predicate R are returned, for that source vertex. This naïve solution of course works as precomputation matches exactly what is needed before the query hits. However, this may not be a feasible solution as geosocial graphs can be very dynamic and are updated all the time. Users check-in all the times at venues and make new friends very frequently. Also such a precomputation is very expensive w.r.t time and may take even days to complete if used on huge graphs. Even if time, $O(V^2)$ is not a concern, it would require quadratic space w.r.t size of the graph, as distances to every possible combination of vertices are stored. Hence this is not a feasible approach.

Another simple idea is to use a Dijkstra's algorithm starting the source vertex and continue until K vertices that fall in the region are found. As Dijkstra's algorithm is greedy it finds the shortest distances in non-decreasing order to all vertices. We can safely conclude that we found the correct solution once we find K vertices in R to be the top-K in that region from the given source vertex. As shown in Figure 10, like in Dijkstra's we start from 's' vertex and start relaxing the neighbors. The marked maroon rectangular box is the region of interest R. 't' and 'x' fall in the

region and say K = 1. From ‘s’ we relax ‘x’ and ‘y’ with 10 and 5 respectively. Then as ‘y’ has the least weight in the priority queue it will be popped and visited. ‘t’, ‘x’ and ‘z’ are relaxed to 8, 14, and 7 as shown in Figure 10 (c). Similarly ‘z’ is relaxed and then ‘t’ is relaxed. Once ‘t’ is popped from the queue, we know its shortest distance has been found and it is 8. Using the parent attribute of each vertex we can find the path to the source.

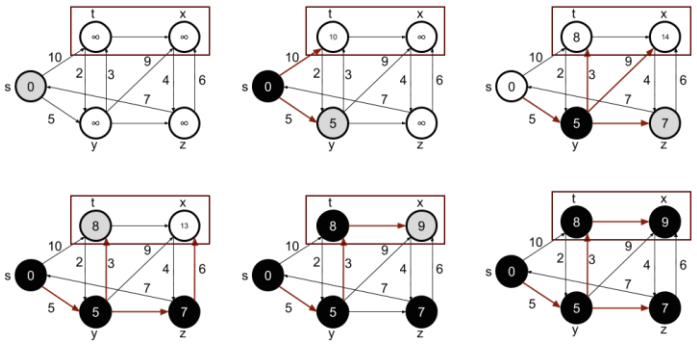

Figure 10. Early stop Dijkstra's Algorithm having a spatial predicate

As only one vertex is needed and ‘t’ is found which OVERLAPS R, and the algorithm halts. If K was set to 2, the algorithm continues until ‘x’ also is also found as shown in Figure 10 (f). The solution is 100% correct due to the way Dijkstra’s reduces the vertices. This approach is feasible as there is not have a huge space requirement. Its biggest plus is no pre-processing and works on highly dynamic graphs. But the side effect is the time it takes to find the solution. A much better solution with a minimal overhead can be invented.

Now that it is understood why existing solutions are not feasible for a real application due to their drawbacks w.r.t time or space, this drives as an inspiration for Socially Proximity & Spatial Reachability (SPSR) index. Referring to Figure 10 again, it can observed that if K = 1, ‘s → y → t’ is the answer. There are two problems here:

1. To understand the 1st problem, assume that the edge z-x doesn’t exist. This means there is no way to reach R from ‘z’. Even then using the above solution ‘z’ is visited first and then ‘t’.
2. The 2nd problem is, vertex ‘z’ is visited before ‘t’ or ‘x’ though it is not part of the final solution even if K = 2.

To solve the first problem SPSR-Spatial is proposed. It helps Dijkstra's during traversal and eliminates routes which do not reach the region, R. This makes Dijkstra's more goal oriented like A*. To solve the second problem another index using the social proximity called the SPSR-Social is proposed. This helps the traversal algorithm, to traverse the graph as less as possible considering both the region of interest and K.

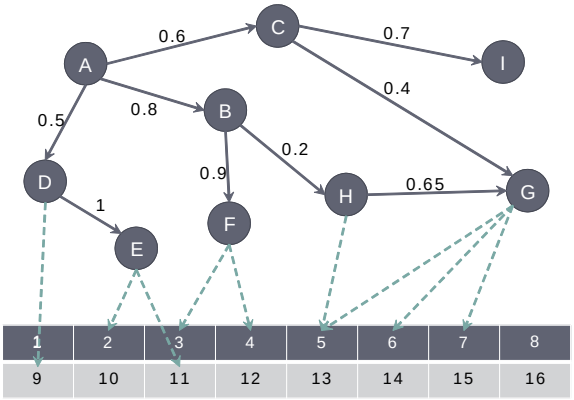

Figure 11. SPSR-Spatial index

The solution is twofold, first data is preprocessed to create an index. Then a modified A* algorithm traverses the graph using the index to answer query of type mentioned in Answering Queries chapter.

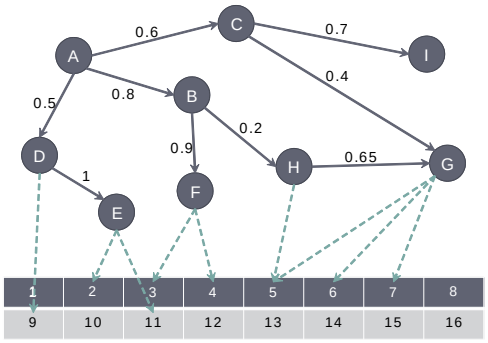

Figure 12. Running Example

A running example is used to explain the main idea followed by a formal algorithm. Consider Figure 2 which has a social graph of friends and their check-ins at various venues.

In Figure 12, each vertex symbolizes a person and two nodes are connected if there is a social relationship between them. The number on the edge indicates their social distance, lesser implies stronger bond. Forest green edges from the nodes to the map are all check-ins made by people

at various venues (nodes not shown), which are our spatial vertices. Forest green edges are also weighted but are not shown as they are not important for explanation purposes.

Our Approach, SPSR

Structure

The goal of the index is to quickly prune the graph for any range predicate, starting at any source vertex. Therefore for every vertex, spatial and social meta information is added. Using this meta information, the traversal algorithm at query time can decide whether to visit a sub-graph starting at that node or not or more generally, this guides the traversal algorithm at query time.

Spatial meta information is created using the spatial at- tributes of the vertices. The world is divided into a fixed number of blocks in space and are numbered in increasing order. Then for each venue (i.e. spatial node) the meta in- formation for all the people nodes (other vertices) who have checked-in there is updated with the block number where the venue belongs. If the world is divided into very fine blocks, each meta entry can be really huge. To compress the index entry, the world is divided again but this time into more coarser blocks and are also numbered like before. For all the index entries which cross a threshold, block numbers from the coarser division are used. This is done recursively until the threshold is satisfied for that index entry. Once this is in place, at query time, while traversing the graph for finding a closest vertex, sub-graphs starting at a vertex which does not reach the region of interest are straight away pruned.

Social meta information/index is created using the social distances (edge weights). A few vertices are picked from the graph based on some criteria (detailed later) and the shortest distances from each to all vertices it can reach are computed using well known single source shortest path algorithms like Dijkstra's and stored. These selected vertices are termed as landmarks. So social meta information is a table which has shortest distances information from each landmark. Using these landmarks and triangle inequality an estimate of the shortest distance from any vertex to any other vertex in the graph is obtained. This valuable information is used during graph traversal to prune sub-graphs even better and is detailed in Answering queries section.

Using the Spatial Component

A complete picture of equally space partitioned world would be as shown in the Figure 3. Here the resolution of the division is 10 by 10. That means the entire world is divided into 10 equal sized blocks horizontally and 10 equal sized blocks vertically. In the running example, the entire region is divided into equal sized blocks from 1 to 16 as shown in Figure 13. Then for vertex D, meta information would be [8] as the user checked-in at a venue which falls in block number 8. Similarly for vertex G, the meta information would be [5, 6, 7]. Continuing like this, a meta information table for each vertex which are directly connected to a spatial node is populated.

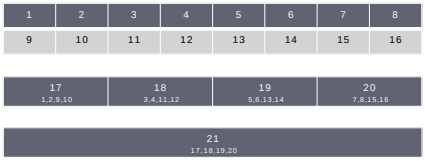

Figure 13. Multilayer grid

This is the reachability to venues by 1-hop which can answer 1-hop queries. For example, did vertex G check-in at venue L1 or did vertex E visit any venue in a region L2. The first can be answered by finding its block number using its location and cross-reference it with the list of blocks G can reach from the meta table. More details on answering queries later.

If idea of 1-hop reachability is extended to any number of hops, it is multi-hop reachability or simply reachability. These values denote all blocks that are reachable from a vertex in any number of hops/steps. For building multi- hop reachability information, direction of all the edges are reversed or the transpose of a graph is created. Now, for every vertex, its reachability information is appended with meta information of all the vertices it is directly connected to bottom up.

In our example after reversing the edges, to compute multi-hop meta information for H, we append G's meta in- formation. Similarly for B's meta information is appended from H and F. As this is a DAG, we have complete reachability information of every node after completing the exercise for entire graph. If the original graph is not a DAG, it is condensed by contracting all strongly connected components into a single node to make it a DAG. The final meta table for every node looks like the one shown in the column 2 of Table 3.

Table 3

Spatial Index

Vertex	Reachable regions	Compressed
D	[9, 2, 11]	[17, 18]
E	[2, 11]	[2, 11]
F	[3, 4]	[3, 4]
H	[5, 6, 7]	[19, 20]
G	[5, 6, 7]	[19, 20]
C	[5, 6, 7]	[19, 20]
B	[3, 4, 5, 6, 7]	[21]
A	[3, 4, 5, 6, 7]	[21]

Using this table we can answer any region reachability query which will be described in the next section. Now we can see that vertex A can reach whatever B, C and D can reach. Similarly B can reach whatever H and F can reach and so on. However, as you can see the size of the meta table (a.k.a index table) can grow really large as vertices like B and A which are highly connected can reach many blocks.

In order to compress the entries in the meta table for highly reachable nodes, a new layer of blocks with higher resolution is added.

As shown in Figure 13 (assume the 3rd layer does not exist for now), blocks 17 to 20 are added on top of blocks 1 to 16. This can be visualized like a stack where the old layer sits exactly on top of the new layer. That is block 17 covers exactly the same area as area covered by blocks 1, 2, 8 and 9. Similarly block 18 in layer 1 represents blocks 3, 4, 10 and 11 of layer 0. Due to this change, the meta table becomes like the one shown in the 3rd column of the Table 3. The reduction factor, which is rate at which the resolution changes between adjacent layers, is set to 1/4 and is a tunable parameter of the system called RF. The number of layers in the multilayer grid index is guided by another system level tunable parameter M. M is the maximum size, in terms of number of block ids, of meta information for a vertex. For example, if M = 2 as used in

the running example, then each vertex can only have two entries in its meta information and until this limit is satisfied, new layers are created. In the example, for all vertices which have more than 2 entries, are compressed using layer 1. However some of them still have more than 2 entries.

To compress further one more layer below layer 1 is added as show in Figure 13. There block 21 represents blocks 17, 18, 19 and 20 in the layer above it.

The discussion began by condensing G to a DAG (G'). Now each node in a strongly connected component gets the meta information of the component as proved Lemma 1 below.

Lemma 1. Let v be a vertex in a strongly connected component C of a directed graph G(V, E). Then,

$$\forall v \in C: \text{Meta information of } v = \text{Meta information of } C$$

Proof. In a strongly connected component C, any vertex can be reached from any vertex by definition, i.e. u reaches v,$\forall(u, v) \in C$.

If C in condensed graph G′ can reach a set of regions R, then any vertex of C can reach R by definition of connected component.

Therefore, meta information of C = meta information of $\forall v \in C$.

Then a R-Tree is also constructed using the spatial nodes in the graph. This will later be used at query time to filter the exact nodes in a region and will be discussed in more details in the next section. These are the two indices created on the spatial component of the graph. For clarity, spatial index always means SPSR-spatial in the rest of the document as discussed in algorithm 1 next.

```
1: function GeoReachPaths-Spatial(G, RF, M)
2:     G' ← G.condense()
3:     for v ← G.V do                          ▷ 1hop spatial reachability
4:         for u ← spatial(G'.Adj(v)) do
5:             v.meta.add(REGION(u))
6:         REPARTITION(v, RF, M)
7:     DO_DFS(G', RF, M)                       ▷ multihop region reachability
8: function REGION(u)
9:     return block # for CURRENT_RES(u)
10: function REPARTITION(v, RF, M)
11:     while v.meta.size() ≤ M do
12:         TRANSLATE_TO_RES(v.meta, CURRENT_RES(v) ÷
    RF)
13: function DO_DFS(G, RF, M)
14:     for v ← G.V do
15:         v.meta.add(DFS_VISIT(G, v))
16:         REPARTITION(v, RF, M)
17: function DFS_VISIT(G, v)
18:     for u ← G.Adj(v) not visited do
19:         v.meta.add(DFS_VISIT(G, u))
20:     return v.meta
```

Figure 14. Algorithm 1, SPSR-Spatial

Algorithm 1 shows how the index, SPSR-spatial is created. In the first phase block numbers of all spatial vertices are added as the meta information of the vertices they are connected from. Then a modified DFS is used to construct multi hop reachability. The REGION() method re- turns the block number for any spatial vertex. The REPARTITION() method ensures the M constraint by recursively increasing the resolution by a factor of RF. In the DFS() method meta information is recursively appended to the head vertex from the tail vertex.

To compute the asymptotic run time and space complexities for algorithm 1, it is divided into four pieces - graph condensation, 1-hop reachability calculation, DFS for multi- hop reachability and repartition function. The runtime for each would be as follows,

- **Graph Condensation:** On line 2, the input graph is condensed into its strongly connected components. Using a popular algorithm like Targan's Algorithm, the runtime would be O(V + E) [19].
- **1-hop reachability calculation:** From lines 3 to 6, 1- hop reachability on the condensed graph is computed. In the worst case, every vertex will be a strongly connected component and so the size of the graph remains the same after graph condensation. As the entire graph is traversed once to populate 1-hop meta data for each vertex, the complexity for this piece also would be O(V + E). Calls to repartition() function are handled separately.
- **DFS:** For multi-hop reachability, a DFS traversal is performed on the graph on line 7. As adjacency list is used for managing the graph's edges, the complexity for DFS would be O(V + E) again. Calls to repartition() function are handled separately.
- **Repartition:** This function is called multiple times to make sure the M constraint is satisfied. The runtime of the function depends on the size of the meta entry for the vertex. For every vertex's index entry, the world is divided with a constant resolution to start with, like 10. The total number of blocks at that default resolution would always be square of it, like 100. Let the total number of blocks in the default resolution be c, which would be the worst case size of any vertex's meta information. This happens when a vertex can reach all blocks in the world. And until the size of meta information for that vertex falls below M,

the resolution of the division is reduced by RF. Therefore the number of times the loop in REPARTITION() function of the algorithm, say n would be,

$$\frac{c}{RF^n} \leq M \quad (5)$$

$$n \leq log_{\frac{1}{RF}} \frac{M}{c} \quad (6)$$

And repartition function is called exactly twice for each vertex, once during 1-hop reachability and once during DFS. Therefore, the time complexity due to this function would be $O(V \times \log_{(1/RF)}(M/c))$. This fraction is very small compared to sum of vertices and edges in the graph.

All the other lines in the algorithm can be computed in constant time. The runtime of SPSR - spatial would hence be O(V + E).

Space complexity would be amount of memory required to store the multi-hop reachability table. Each index entry has an upper bound of M. Hence memory consumption in the worst case would be, $O(V \times M)$.

Using the Social Component

Social distances between nodes are used to create an additional index to prune the graph even better. This will take care of the cases when the graph is very dense and the spatial index created before may not be of much use for pruning at query time. More details on querying the graph are described in the next section.

The main idea is to select a few nodes in the graph and call them landmarks. Then for each vertex shortest distances to each landmark is stored. Then at query time these precomputed distances and triangle inequality are used to guide as a heuristic in the A* search algorithm. The inspiration is from [8] which introduces a class of algorithms called ALT. The main challenge here however is to find top-k closest vertices to a given vertex in a region and not finding the shortest path from a given source to a given destination unlike in [8].

The quality of the landmarks determine the pruning power of the index. Choosing the right landmarks requires some domain knowledge of the graph. Once that is picked the process remains the same no matter what the graph represents. [8] talks about multiple ideas on how to find high quality landmarks quickly. The ideal case would be to find as minimum number of

landmarks as possible such that every vertex in the graph is connected to at least one of the landmarks. However leaving out a few vertices that do not reach any landmarks will not hamper the correctness of the algorithm. Therefore finding a sweet spot of number of land- marks which gives the best query performance is crucial and is the main goal of [8]. The main contribution from our side is to use this social index and propose a new heuristic for A* algorithm that finds top-k venues satisfying a spatial predicate.

```
1: function GeoReachPaths-Social(G)
2:     L ← FIND_LANDMARKS(G)                    ▷ Domain based
3:     social_index ← []
4:     for l ← L do
5:         for v ← G.v do
6:             social_index[l][v] ← shortest distance from l to v
```

Figure 15. Algorithm 2, SPSR-Social

Algorithm 2, SPSR-Social, describes how to create an index using landmarks. Landmark selection function on line 2 can be any of the functions described in [8]. Then for each landmark the shortest distances is computed using any well-known single source shortest path algorithms like Dijkstra's or Bellman Ford to every vertex reachable from that landmark. Please note that distances from the landmark to every vertex are saved and not the other way around. The direction is important as we are only dealing with directed graphs.

To compute asymptotic run time and space complexities, algorithm 2 is divided into two pieces - finding the land- marks, finding shortest distances to each reachable vertex from each landmark. The runtime for each is as follows,

- **Finding Landmarks:** There are various ways of picking the landmarks and is totally left to user. In our case we used an approach which finds landmarks in constant number of scans of the entire graph. Therefore, the complexity of this piece is O(V + E).
- **Shortest distance to each landmark:** Here, the shortest distances from each landmark to all vertices it can reach are saved. Using adjacency list for storing the edges, Dijkstra's algorithm is used as edges have positive weights. With this setting, the complexity would be O(landmarks × E log V). As the number of landmarks is usually very small compared to number of edges in the graph, it would be O(E log V) in asymptotic notation.

Therefore the runtime for this algorithm using a sensible landmark selection algorithm would be, O(V + E) + O(E log V).

Space complexity would be memory taken to store the shortest distances to each reachable vertex for all landmarks. As the number of landmarks is very small compared to the number of vertices in the graph, this would be O(V).

Answering Social Proximity & Spatial Reachability Queries

A modified A* with landmark [8] algorithm is proposed for answering SPSR queries using the SPSR index. The main goal is to prune as much graph as possible using the spatial index and move in a goal oriented manner towards the region during traversal. The algorithm takes a graph G, a starting vertex s and a query rectangle R as input and returns the top-K vertices by social distance in R in an iterative manner. Being iterative helps pipeline the SPSR with other database functions.

The crux of A* algorithm is the heuristic function. Our heuristic function takes a vertex and a region and returns the heuristic distance which will be used by A* to decide which path to traverse. In order to design such heuristic function SPSR-social index is used explained in algorithm 3. For this the triangle inequality property is used to get a lower bound on the distance between any two vertices.

```
1: function RRP(G, s, R, K)
2:     nearest_vertices ← []
3:     Q ← [s]
4:     best_index ← 0
5:     while Q is not empty do
6:         v ← EXTRACT_MIN(Q)
7:         if v lies in R then
8:             nearest_vertices << v                    ▷ add v to set
9:             best_index ← best_index + 1
10:            if nearest_vertices.length = K then
11:                return nearest_vertices
12:            for v ← Q do      ▷ Update heuristic for existing in Q
13:                v.key ← v.d+ HEURISTIC(u, R, best_index)
14:        if v is not visited then
15:            VSIT(G, u, v, Q)
16: function HEURISTIC(u, R, i)
17:     v ← RTREE.filter(R)                          ▷ vertices in R
18:     return MAX( MIN_i(GRP-SOCIAL(l, u) ∀u ∈ v)∀l ∈ L)
```

Figure 16. Algorithm 3, SPSR

```
1: function VISIT(G, u, v, Q)
2:     for u ← G.Adj(v) do
3:         if R does not lie in GRP-SPATIAL(u) then
4:             continue
5:         if u is not visited then
6:             if u in Q then
7:                 if u.d < v.d + WEIGHT(u, v) then
8:                     continue
9:                 else
10:                    u.key ← v.d + WEIGHT(u, v)
11:            else
12:                u.parent ← v
13:                u.d ← v.d + WEIGHT(u, v)
14:                u.key ← HEURISTIC(u, R, best_index)
15:                Q.INSERT(u)
```

Figure 17. Algorithm 4, Vertex Visit sub routine for algorithm 3

In Figure 18, say H and R vertices need a lower bound on the distance between them. For this, the distances saved w.r.t. each landmark, here G as part of social SPSR index is used. So in the figure using the index the distances u and v are known. In order to find x which is the distance between H and X u is subtracted from v. Therefor the value of x shown in the figure would be v − u which directly follows from vector addition. This is only a lower bound on the distance from H to X and is proved in [8].

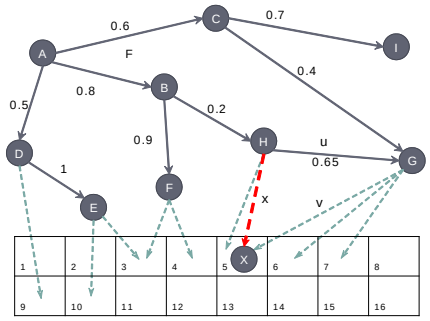

Figure 18. Triangle Inequality

In our case a lower bound from a vertex to a region is needed. In order to understand how this is done, recap the problem definition - find top-k closest vertices (w.r.t social distances) in a region from a vertex in a graph. To under- stand better, set K = 1, i.e. say the nearest vertex in the R from a source vertex is needed and say we have only one landmark. Now, for A* to work efficiently, a lower bound as tight as possible is needed, else the traversal would touch as many vertices as Dijkstra's. Let the source vertex be H, landmark be G and region exactly enclosing blocks 5 and 6 in the Figure 5. The problem now becomes finding the closest vertex to H in the region. To get a lower bound for x, the triangle inequality formula devised above, v − u is used. In

order to keep x minimum, v should be as small as possible. Therefore the closest vertex in R to G is picked. Similarly for the second closest vertex we choose the second nearest vertex to G in R. The algorithm proceeds this way till the nearest K vertices in R are found. If multiple landmarks are present, a value of x for each landmark is produced. We pick the maximum value of x to get the tightest bound possible and this follows from efficiency of A* algorithm.

Algorithm 3 shows Social Proximity & Spatial Reachability in detail. All vertices are labelled as unvisited initially. The visited flag is used to prevent traversing the same node multiple times. The priority queue Q in keyed by sum of actual distance from source and a heuristic distance to current closest vertex to a landmark. For each vertex popped from the queue, it is tested if it OVERLAPS R and returned if K vertices are found. If not, keys for all existing vertices in Q are updated with the new heuristic. All unvisited neighbors of the popped vertex are enqueued only if they can reach the region R as per SPSR-Spatial index. If a neighbor can reach R, and if it not already in Q, its heuristic distance is computed and summed with the distance from the source and inserted into the Q. If the vertex is already in the Q, its key is updated if the new distance is smaller. The algorithm proceeds this way till all vertices in the Q are exhausted or K closest vertices in R are found whichever is earlier. This way, it is an iterative algorithm which doesn't traverse the entire graph to return the top-K results.

Algorithm 3 answers socio-spatial queries using modified A* with landmark algorithm and its complexity depends on the quality of the heuristic function. So rather than one value for asymptotic runtime two extremes are obtained. The algorithm can be divided into three pieces - the Q, heuristic function and vertex visit (which is written as a subroutine as algorithm 4). The runtime for each is as follows,

- **the Q and Vertex visit:** Lines 5 to 15 of algorithm 3 detail the priority queue's role (viz. keyed by actual distance + heuristic distance). Algorithm 4 details what happens at every vertex that is not yet visited. The number of times the Q loop executes depends on the way heuristic guides the algorithm. In the best case, it is always on the right path to the current shortest distance and so the run time would O(n), where n is the length of the path. As K such paths are needed, the complexity would become, O(K × n). In the worst

case, the heuristic always picks the wrong path and the algorithm works like a Dijkstra's or a BFS. In such a case, the complexity would be O(V + E) for finding any number of shortest paths since the entire graph is traversed once.

- **the Heuristic:** Here all vertices that fall in the region of interest, R are filtered. If properly implemented this can be done only once per query. Its complexity would be $O(\log_m(V))$ where m is the number of nodes/vertices per memory page (fan out of a tree). Then the maximum of all closest distances to all landmarks if found. This is nothing but finding the K smallest elements in an array, as the number of landmarks are constant. Its complexity would be $O(K + (V - K) \log K)$. So the total complexity would be $O(\log_m(V)) + O(K + (V - K) \log K)$ where m is the number of nodes/vertices per memory page (fan out of a tree).

Therefore the runtime of Social Proximity & Spatial Reachability would be in between $O(K \times n) + O(\log_m(V)) + O(K + (V - K) \log K)$ and $O(V + E) + O(\log_m(V)) + O(K + (V - K) \log K)$, where m is the number of nodes/vertices per memory page.

CHAPTER 5

EXPERIMENTS

In this chapter, multiple experiments verify the SPSR algorithm with various input parameters. For this an Intel Core i7 2.66 GHz processor with 8GB RAM and running MAC OSX was used.

Real Yelp Dataset[3] which has both social and spatial components as introduced in the beginning was used. The dataset has 552K social nodes, 77K spatial nodes, 3.5M social edges and 2.2M spatial edges. As social edges indicate friendship strength between users, a random number between 1 to 10 was generated to signify the social distance between two users. Similarly, as every spatial edge is a check-in at a business, the rating given by the user was indicated by a random number between 1 and 10. In both cases larger the number, lower is the friendship strength and lower is the rating respectively.

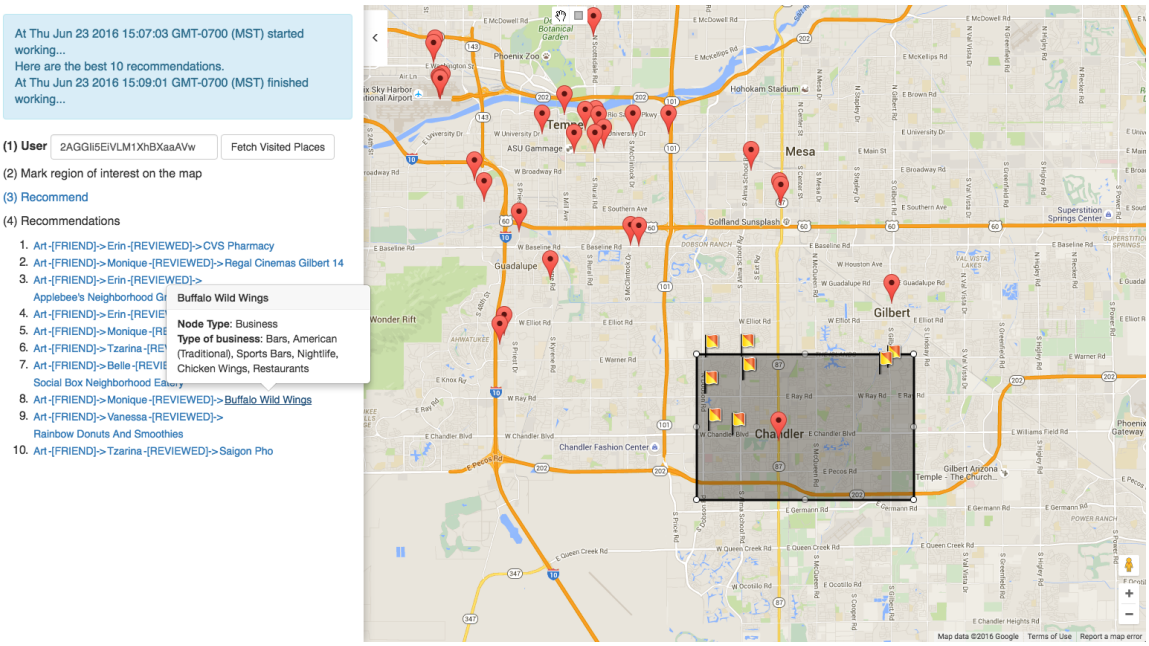

Figure 19. Screenshot of the web application showing the top 10 venues in a region

The closest vertices returned by SPSR are fed into a web application which visualizes the results as shown in Figure 19. The red markers are the places which the user,

[3] https://www.yelp.com/dataset_challenge

“2AGGIi5EiVLM1XhBXaaAVw” visited and flag markers are her recommendations based on social distances. All the recommendations are for the region marked by the translucent black rectangle. The figure also shows the shortest paths listed in ascending order by social distances. Hovering over a node in the path, shows all the information rich attributes that can be used for computing social distance (or edge weights).

First the user name is entered and the list of places the user been to is listed using the left panel. This gives an idea where the user usually goes to. Then a region is marked using the rectangular marquee tool on the top of the map. Lastly the Recommend button fetches the best 10 recommendations (10 closest venues to user) in the marked region on the map. The details of the application and its source code are available at

https://github.com/Nithanaroy/GeoReachRecommender.

Table 4

Default Parameter Values

Parameter	Default Value	Range
K	100	10, 100, 1000, 10000
RZ	125	25, 125, 625, 3125
VertexReachesAlgo	Type 3	Type 1, Type 2, Type 3
M	10K	-
RF	4	-

Unless specified each run uses the default parameter values as shown in Table 4. Parameter K is the shorthand for top-k, i.e. any query requests 100 closest vertices by default. RZ is the shorthand for resolution which deter- mines the number of blocks the world is divided into. RZ ← 125 implies that the world is equally into 125 by 125 blocks along latitudes and longitudes. VertexReachesAlgo parameter defines how line 3 of algorithm 4 is implemented. Throughout this section, it is referred as OVERLAPS. Three implementations which check whether the query region R overlaps with regions reachable from a vertex were used and will be described near those experiments. M and RF are the same parameters described before as the maximum

allowed size of a SPSR-Spatial index entry for a vertex and reduction factor between levels of the SPSR-Spatial index respectively.

The pre-processing times for building the SPSR-spatial and SPSR-social indices are very reasonable for a dataset of realistic size dealt here. The pre-processing time for SPSR is summation of times taken for building both the social and spatial indices which is 195s. SocialFirst does not require any pre-processing as it is a modification of Dijkstra's algorithm. SpatialFirst requires selection of landmarks and finding shortest distances for each which is exactly like the social index of SPSR.

Table 5

Pre-processing times

SPSR	Social First	Spatial First
SPSR-Spatial requires 122s SPSR-Social requires 73s	0s	73s

SPSR is first compared with SocialFirst and SpatialFirst approaches introduced in Preliminary chapter. Figure 20 compares the time taken by SPSR, SpatialFirst and SocialFirst algorithms for the same source vertex and region. It may seem very intuitive at first that the SpatialFirst algorithm should to- tally beat others as pruning the graph by a huge extent initially using r-tree. To be exact from a graph of 629K nodes, we focused on 2,804 nodes only which is 0.44% of the entire graph. Surprisingly, SpatialFirst is 3 orders of magnitude slower than the simple graph traversal Social- First algorithm and 4 orders of magnitude slower than SPSR. The reason for this massive difference is, A* with landmark is designed for solving single source and single destination problems. In our case A* with landmark function is invoked from the single given source vertex to every destination vertex in the region. Getting into some digits, say A* with landmark takes 3s (which is a very modest number on a real Yelp graph) for computing the shortest path for a given source and destination. Assume the given region R contains 2,000 venues. Therefore A* with landmark is invoked to each of them for a total of 2,000 times, which itself takes about 6,000s or 1.67 hours! Another way to visualize is, the algorithm (from source) is

restarted for each of 2,000 destinations which causes it to lose by a huge margin with SocialFirst and SPSR.

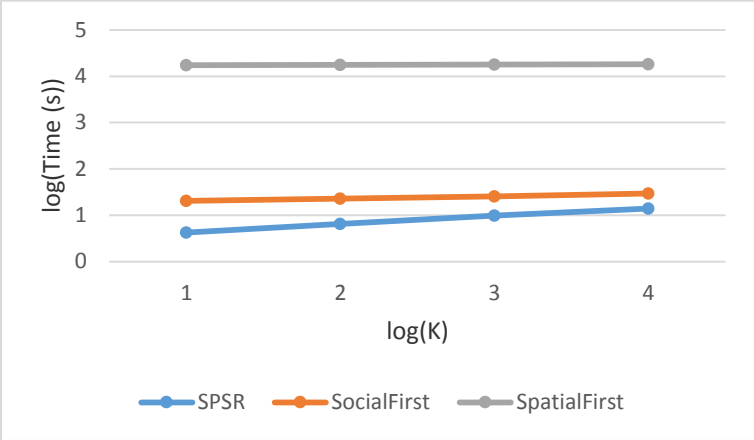

Figure 20. Algorithms VS K VS Time

That is where the SocialFirst algorithm shines. It totally takes a disconnected approach between spatial and social constraints like SpatialFirst, however it doesn't restart for obtaining the next shortest path. SpatialFirst aimlessly wanders the graph in the order of increasing distances from source and emits a result when it finds a vertex in the region. SPSR is the best of both the worlds, as it uses a heuristic to traverse the graph in a goal oriented manner towards the region like SpatialFirst and does not restart for obtaining the next shortest destination like SocialFirst. This is the main reason why it shines than the other approaches. And this can be clearly seen in Figure 20. As SpatialFirst is way out of the league, it is eliminated from the discussion from now on and SocialFirst and SPSR are focused. From the plot it is evident that as K increases, the time taken by SPSR and SocialFirst also increases. Though it may be very unlikely that for a query to seek with K > 20, SPSR still outperforms SocialFirst algorithm by at least 2 times even in the worst cases. To be specific, the region in the query has 2,804 spatial nodes and K was set as high as 10,000 nodes which is 100% selectivity for that region and is testing the limits. As the final outcome is clear, the main focus would now be only on SPSR. As SPSR is made of Spatial Index and Social Index, SPSR was run multiple times with and without each index and to study how each of them perform.

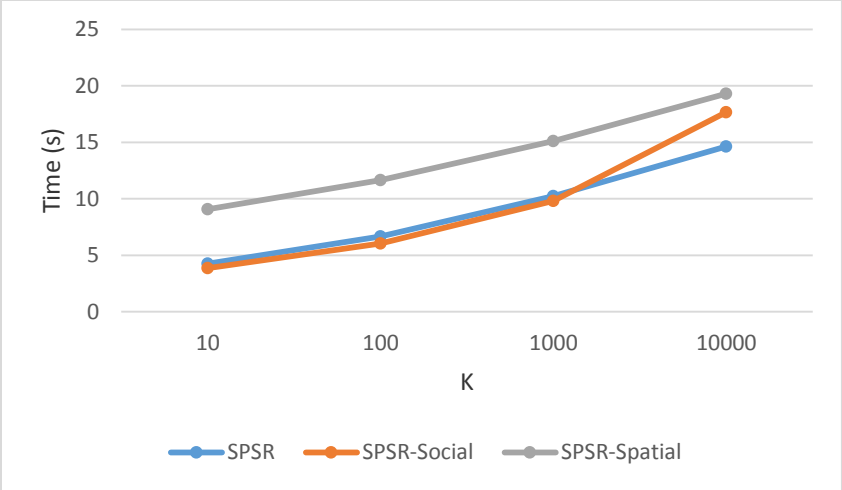

Figure 21. Time VS K VS SPSR Types for Resolution = 25

So the Figure 21 breaks down the components of SPSR into SPSR-Spatial, SPSR-Social and Both (marked as SPSR in the plot's legend). SPSR-Spatial uses only the Spatial index while traversing the graph. As using only SPSR-Spatial index, a heuristic function cannot be built, modified Dijkstra's is used with one extra condition. In Dijkstra's, before adding any vertex to the priority queue, it is verified whether it can reach the given region R using the spatial index. So the sub graphs which cannot reach the region are pruned early using the index. Similarly, SPSR-Social uses only the social index for finding the K closest vertices to the source. For this, the algorithm is exactly like algorithm 3 except the condition on line 3 of algorithm 4 would always return **false**. This indirectly means spatial index is never used.

Though using either of the indices beat SocialFirst and of course SpatialFirst algorithms, it can be clearly seen that using SPSR-Social index outperforms others for smaller K in this case. To understand why it is so, revisit the heuristic algorithm of SPSR. The heuristic function uses the Social index with landmark(s). Combined with triangle inequality a lower bound on the distance between the source vertex and a destination vertex in the region is computed. Higher the lower bound, better is the pruning power of SPSR. As dis- cussed the quality of landmark(s) plays an important role in the performance. So here using the spatial index only adds the overhead by querying that index, therefore the curve using both the indices slightly underperforms initially. But as K increases, the heuristic's bound weakens as v in Figure 18 is no longer small compared to u.

At the same time, spatial index helps SPSR to make better decisions whenever social index fails. The bottom line, using social index gives better results when:

- Quality of the heuristic, indirectly landmarks, is very good
- Graph is very dense that most of the vertices can reach the region, making spatial index only an overhead

However, as the value of K increases, using both the indices certainly helps as unnecessary graph traversals are further reduced by spatial index. This is clearer in a later experiment when the quality of the landmark is not as good as this case. A very low resolution of 25 by 25 was used above. So how the runtimes change when we increase it by 125 times is studied next.

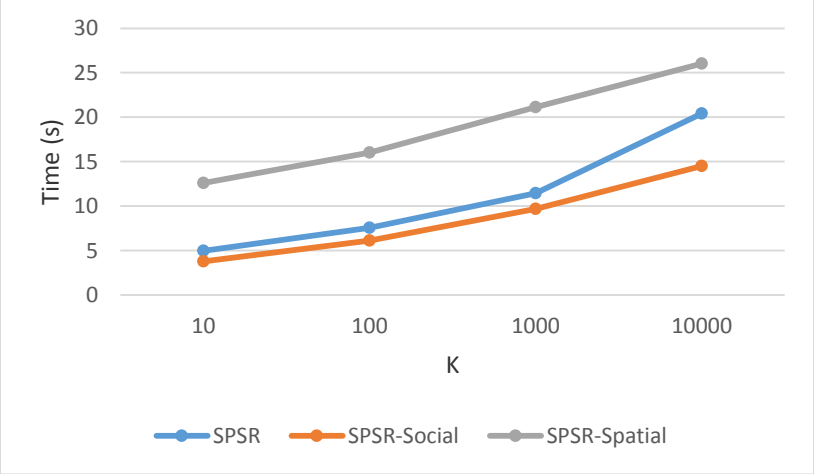

Figure 22. SPSR Types VS Time VS K for Resolution = 3125

Just as expected as shown in Figure 22, the gap between SPSR and SPSR-Social indices further increases when resolution (RZ) is set to 3125. The function OVERLAPS which checks whether a vertex can reach a region using spatial index takes longer when size of the index entry increases for a given vertex. To totally confirm that this is the case, OVERLAPS was implemented in two more ways which were equivalent w.r.t. runtime but cash on tiny advantages based on the size of R and size of spatial index. This is exactly the same parameter VertexReachesAlgo in Table 4. This is how each algorithm is implemented:

- **Type 1:** It is a No or May Be algorithm. In this axes transformation technique is used to check if a vertex reaches a region. The axes are transformed such that the southwest corner of the region in the query, is set as origin (0,0). Then for checking if a vertex

reaches this region, SPSR-Spatial entry for the vertex is queried. Then each block number, that the vertex can reach as per reachable blocks table, is transformed into the new co-ordinate system. Then using this transformed 2D block number, a simple comparison of the block with all four corners of the region was used. If the result is true, the vertex may reach R and so we have to use the exact Type 2 or Type 3 algorithms.

- **Type 2:** In this, region (R) in the query is enumerated to block numbers for as many levels as there in the spatial index. Then to check if a vertex reaches this region, each block number from the spatial index for the vertex, are probed with the enumerated region (R). This probing can be done in constant time if blocks reachable by a vertex are stored in a HashSet. This approach performs better than Type 1 if region is really small as the overhead of transforming into a new co-ordinate system is avoided.
- **Type 3:** In this also, region (R) in the query is enumerated to block numbers for all levels in the spatial index. Then a native set intersection function to find if there is match between blocks from R and blocks from a vertex was used. This sometimes shines as native programming implementations which are written in most optimized way especially in higher level languages like Python.

That is how each of the three OVERLAPS algorithms are implemented. Figures Figure 23, Figure 24, Figure 25 clearly show that all perform the same way. However, if on mashing the plots together keeping the K constant, Algorithm of Type 3 outperforms in majority of the cases. This confirms the previous doubt that if the resolution is too high like 3125 by 3125 the overhead in OVERLAPS function overcomes the advantage gained by graph pruning in a dense graph. Now to figure out the sweet spot for right value of RZ, experiments comparing K VS Resolution VS Time for each variant of SPSR were studied.

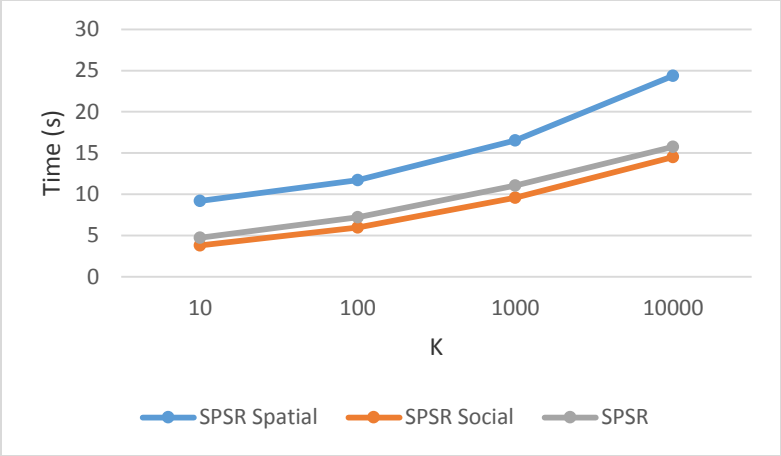

Figure 23. K VS Time VS SPSR Types for VertexReachesAlgo Type 1 and Resolution = 625

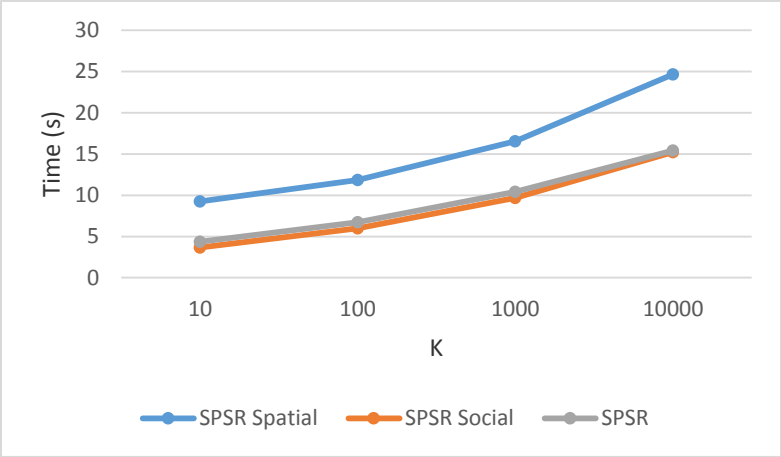

Figure 24. K VS Time VS SPSR Types for VertexReachesAlgo Type 2 and Resolution = 625

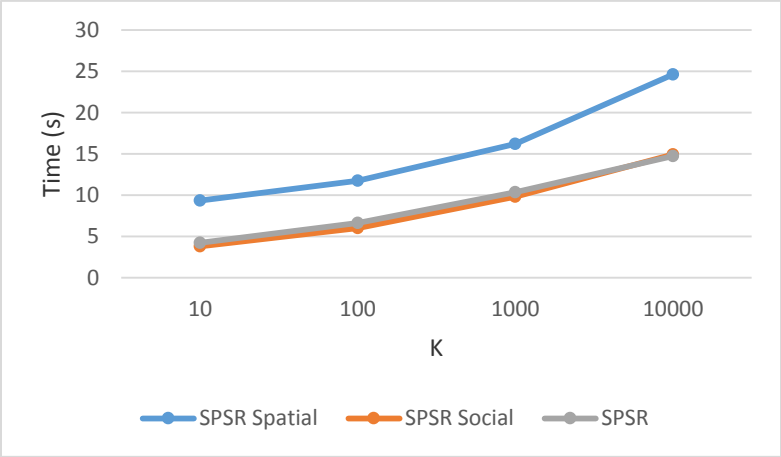

Figure 25. K VS Time VS SPSR Types for VertexReachesAlgo Type 3 and Resolution = 625

From Figure 26 it is evident that as resolution increases for a fixed K and for SPSR-Spatial, performance degrades for very high resolutions due to the overhead by OVERLAPS function. For very low resolutions, as each block is almost the size of Texas, even if a user checks-in at one restaurant there, he/she is considered reachable to that block. So it returns that most vertices can reach R, making it less useful to use a spatial index. The sweet spot so is in between the both extremes, which is 625 in this case. Similar conclusions can be made in next case Figure 27 where both indices were used.

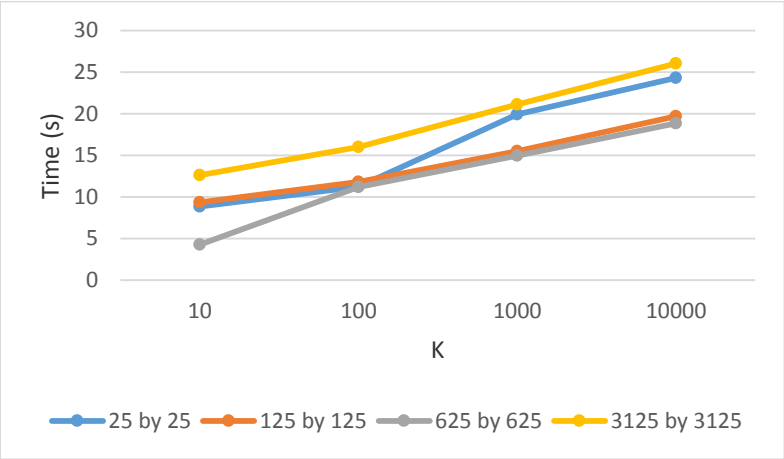

Figure 26. Time VS K VS Resolution for SPSR-Spatial Algorithm

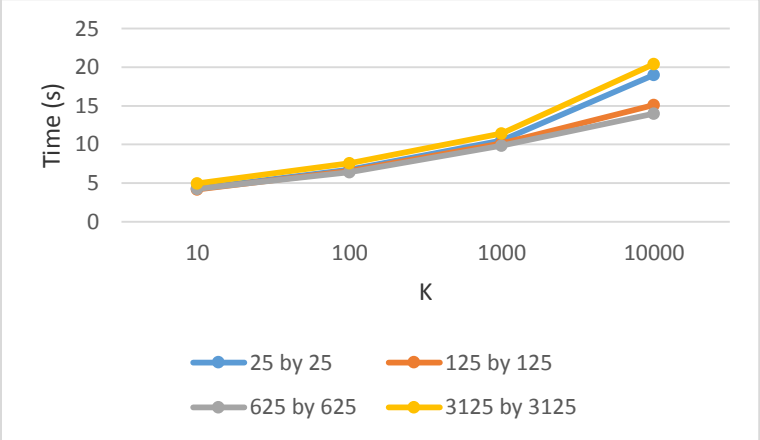

Figure 27. Time VS K VS Resolution for SPSR Algorithm

But social index lessens the loss brought by spatial index overhead and therefore extreme high resolutions also per- form at par with lesser resolutions for smaller K. For larger K even quality of heuristic goes down, so the gap widens.

So after these experiments, it can be concluded that when the graph is really dense using the social index with a high quality landmark is sufficient. Things change when the quality of landmark(s) is not as good. For the next query, the region is even more densely connected and the source vertex is the same, however a lower quality landmark was used. This region has 21,239 spatial nodes which is almost 10 times the count of the previous one.

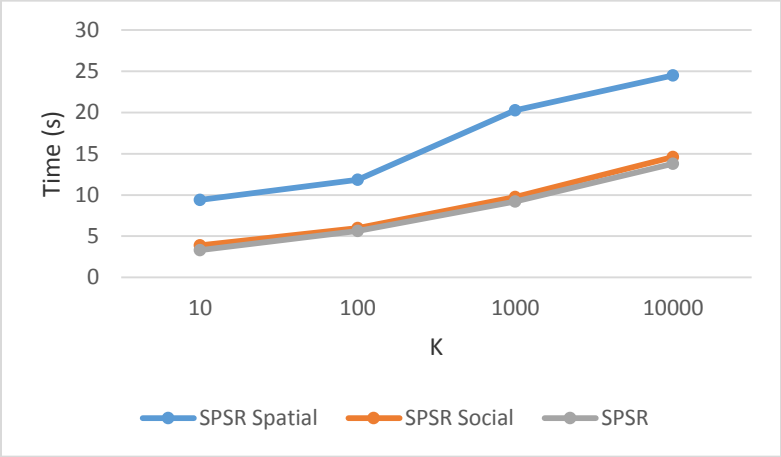

Figure 28. Runtime comparison between the types of SPSR algorithms for RZ = 625 ***and for lower quality landmark***

Figure 28 proves why just having a social index won't help like before. For this region, purposefully a lower quality landmark was chosen. In such cases spatial index prunes majority of the graph as a good resolution of 625 by 625 found earlier was used. Though social index equally performed between Resolution and K using tween Resolution and K using social + spatial index initially, it lost soon as the landmark quality further degraded for higher K for the same reason explained before. The correctness is never compromised, it is only that SPSR tends to Dijkstra's search if landmark quality is not good. Now that when to use each type of index is understood, how each of the algorithms perform with change in region size is studied next.

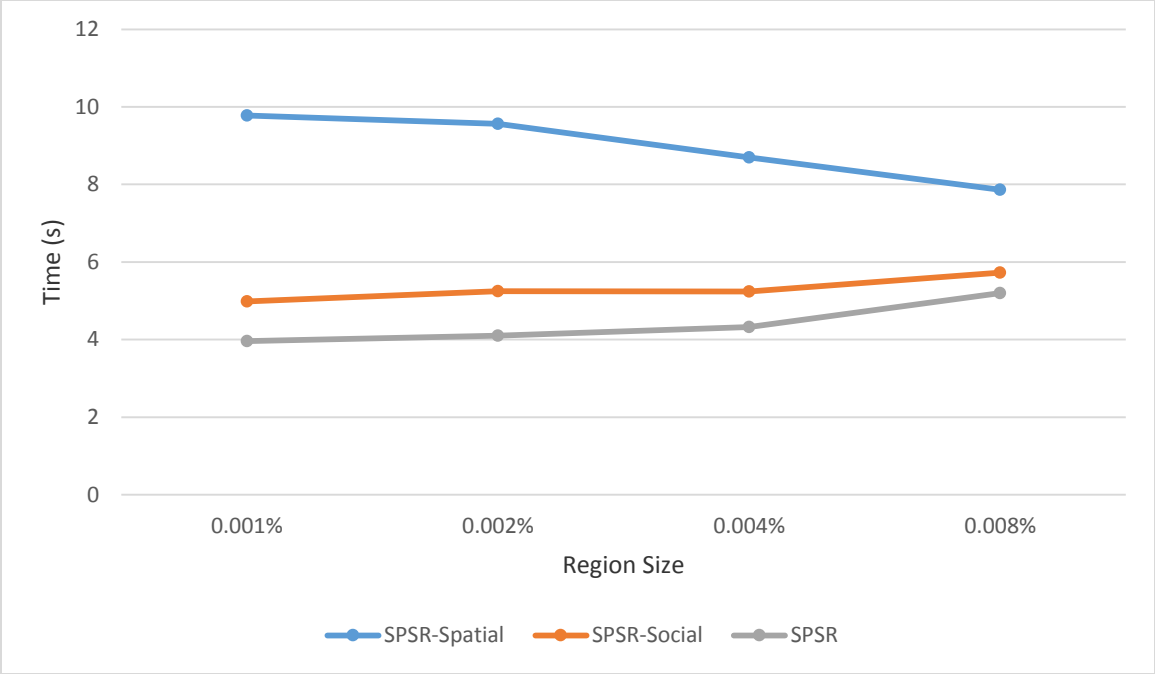

Figure 29. Region Size VS Time VS SPSR Type for source vertex S1

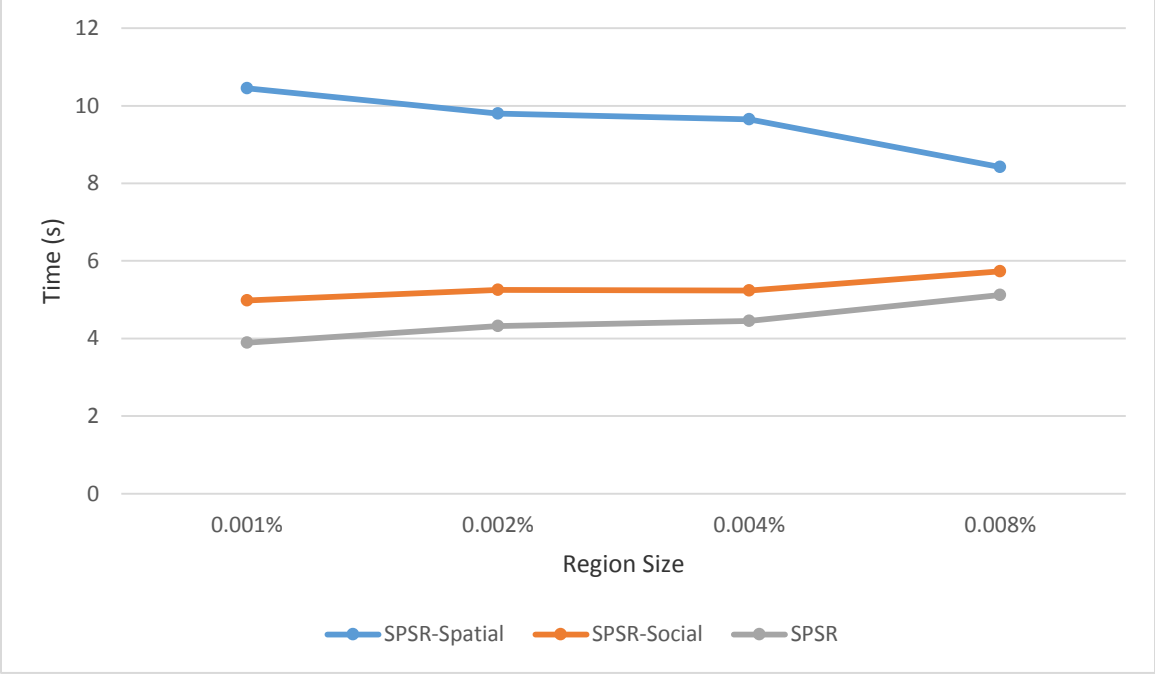

Figure 30. Region Size VS Time VS SPSR Type for source vertex S2

Figure 29 shows how in every algorithm the time taken linearly increase w.r.t. the size of the region. Region of size 0.001% implies, 0.001% of area of the entire world. In Figures Figure 29, Figure 30, in both the cases users near the source vertex have many check- ins in the given

regions and so the spatial index performs worse initially due to its overhead but gradually performs better. The gap further reduces when we a user whose social neighbors do not have many check-ins in the query regions was chosen. Due to this algorithm has to traverse the social graph further down to find the result as shown in Figure 30.

CHAPTER 6

CONCLUSION

SPSR finds socially k-NN with spatial range filter by combining social and spatial searches and works on any socio-spatial graph. Different tunable parameters give an extra layer of flexibility to such a generic solution. Thorough experiments not only prove this point by outperforming existing approaches by at least three times even in extreme cases but also show how to set each of the tunable arguments. Extensions to SPSR can include a persistent way to store the index and also a distributed algorithm.

REFERENCES

[1] Nikos Armenatzoglou, Stavros Papadopoulos, and Dimitris Papadias. A general framework for geo-social query processing. Proceedings of the VLDB Endowment, 6(10):913–924, 2013.

[2] Frank M Bass. A new product growth model for consumer durables. manageniient sci. 15 215-227.. 1980. the relationship between diffusion rates, experience curves, and demand consumer durable technological 53:51–67, 1969.

[12] Matthew Richardson and Pedro Domingos. Mining knowledge-sharing sites for viral marketing. In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 61–70. ACM, 2002.

[13] Philippe Rigaux, Michel Scholl, and Agnes Voisard. Spatial databases: with application to GIS. Morgan Kaufmann, 2001.

[14] Hanan Samet. Foundations of multidimensional and metric data structures. Morgan Kaufmann, 2006.

[15] EdunovCarlos Sergey, Onur F DiukIsmail, and Burke BhagatMoira. Three and a half degrees of separation. https://research.facebook.com/blog/ three-and-a-half-degrees-of-separation/.

[16] Shashi Shekhar and Sanjay Chawla. Spatial databases: a tour, volume 2003. prentice hall Upper Saddle River, NJ, 2003.

[17] S Skiena. Dijkstra's algorithm. Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley, pages 225–227, 1990.

[18] Yuhan Sun and Mohamed Sarwat. Georeach: An efficient approach for evaluating graph reachability queries with spatial range predicates. arXiv preprint arXiv:1603.05355, 2016.

[19] Robert Tarjan. Depth-first search and linear graph algorithms. SIAM journal on computing, 1(2):146–160, 1972.

[20] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pages 845–856. ACM, 2007.

[21] Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on, pages 75–75. IEEE, 2006.

[22] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. Grail: Scalable reachability index for large graphs. PVLDB, 3(1-2):276–284, 2010.

[3] Richard Bellman. On a routing report, DTIC Document, 1956.

[4] Jacqueline Johnson Brown and elasticities for innovations. J. Bus, problem. Technical Peter H Reingen. Social ties and word-of-mouth referral behavior. Journal of Consumer research, 14(3):350–362, 1987.

[5] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. SIAM Journal on Computing, 32(5):1338–1355, 2003.

[6] Pedro Domingos and Matt Richardson. Mining the network value of customers. In Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, pages 57–66. ACM, 2001.

[7] Eyal Even-Dar and Asaf Shapira. A note on maximizing the spread of influence in social networks. In International Workshop on Web and Internet Economics, pages 281–286. Springer, 2007.

[8] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[9] M Helft. Bing taps facebook data for fight with google, 2011.

[10] David Kempe, Jon Kleinberg, and E´va Tardos. Maximizing the spread of influence through a social network. In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 137–146. ACM, 2003.

[11] Kyriakos Mouratidis, Jing Li, Yu Tang, and Nikos Mamoulis. Joint search by social and spatial proximity. Knowledge and Data Engineering, IEEE Transactions on, 27(3):781–793, 2015.

[23] Cormen, T. H. (2009). Introduction to algorithms. MIT press.

[24] Bao, J., Zheng, Y., & Mokbel, M. F. (2012, November). Location-based and preference-aware recommendation using sparse geo-social networking data. In Proceedings of the 20th international conference on advances in geographic information systems (pp. 199-208). ACM.

[25] Zheng, Y. (2011). Location-based social networks: Users. Computing with Spatial Trajectories, Yu Zheng and Xiaofang Zhou, Eds.

[26] Li, Q., Zheng, Y., Xie, X., Chen, Y., Liu, W., & Ma, W. Y. (2008, November). Mining user similarity based on location history. In Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems (p. 34). ACM.

[27] Xiao, X., Zheng, Y., Luo, Q., & Xie, X. (2010, November). Finding similar users using category-based location history. In Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems (pp. 442-445). ACM.

[28] Xiao, X., Zheng, Y., Luo, Q., & Xie, X. (2014). Inferring social ties between users with human location history. Journal of Ambient Intelligence and Humanized Computing, 5(1), 3-19.

[29] Zheng, Y., Zhang, L., Ma, Z., Xie, X., & Ma, W. Y. (2011). Recommending friends and locations based on individual location history. ACM Transactions on the Web (TWEB), 5(1), 5.